Solidity Parsing Using SmaCC: Challenges and Irregularities

Henrique Rocha Stephane Ducasse
Marcus Denker
INRIA Lille - Nord Europe
{henrique.rocha, stephane.ducasse,
marcus.denker}@inria.fr

Jason Lecerf
CEA-List
jason.clement.lecerf@gmail.com

Abstract

Solidity is a language used to implement smart contracts on a blockchain platform. Since its initial conception in 2014, Solidity has evolved into one of the major languages for the Ethereum platform as well as other blockchain technologies. Due to its popularity, there are many tools specifically designed to handle smart contracts written in Solidity. However, there is a lack of tools for Pharo to handle Solidity contracts. Therefore, we implemented a parser using SmaCC to serve as a base for further developing Solidity support in Pharo. In this paper we describe the parser creation, the irregularities we found in the Solidity grammar specification, and common practices on how to adapt the grammar to an LR type parser. Our experiences with parsing the Solidity language using SmaCC may help other developers trying to convert similar grammars.

Keywords Solidity, Parser, SmaCC, Blockchain, Ethereum.

1. Introduction

The Blockchain technology attracted a lot attention recently [LCO+16]. Blockchain is a distributed database, managed by a peer-to-peer network that stores a list of blocks or records. Ethereum [Eth14], and BitCoin [Nak09] are examples of blockchain technologies. Blockchains can be used for many applications such as cryptocurrency, digital wallets, adhoc networks, and remote transactions [LCO+16, HL16, LMH16, Dzi15, Eth14, Nak09]. One notable application of blockchain is the execution of smart contracts [LCO+16].

Smart contracts are what embedded procedures are for databases: programs executed in the blockchain to manage and transfer digital assets. When used in platforms like Ethereum, the contract language is Turing-complete [BDLF+16]. Therefore, smart contracts can be used in many different scenarios [LCO+16]. For example, there are smart contracts employed to simple storage [Eth17], and outsourced computation [LTKS15].

Solidity [Eth17] is a programming language loosely based on JavaScript, and it is used to specify smart contracts on blockchain platforms. Solidity was originally designed to be the primary smart contract language for the Ethereum platform. Even though other contract languages have been created for Ethereum [DAKM15], Solidity is still one of the major ones. Moreover, Solidity can also be used in other blockchain platforms such as Monax[1] and Hyperledger[2].

Probably because of its popularity, there are many tools to help integrate smart contracts written in Solidity with other languages and technologies [Eth17]. For example, we have Solidity compilers coded in C/C++ and NodeJs, third-party parsers and grammar specifications (JavaScript and ANTLR), plugins for IDEs and editors (IntelliJ, Visual Studio, Vim, Atom, and etc.). Such tool integration support developers of smart contracts. However, as far as we know, there is a lack of tools for Pharo Smalltalk to handle Solidity smart contracts. Moreover, most academic work towards smart contracts focuses on security [LCO+16, BDLF+16, DAK+15] and not in tool support.

In this paper, we plan to partially tackle this lack of tools problem by building a Solidity parser that runs on Pharo Smalltalk. We claim that with a parser and its generated AST (Abstract Syntax Tree), we will be able to develop strong tool support for Solidity contracts. For instance, it would be much easier to create code inspection tools on top of a functional parser than to rely on the purely textual content of the contract. To accomplish these goals, we used SmaCC

[1] https://monax.io/, verified 2017-06-19.

[2] https://www.hyperledger.org/, verified 2017-06-19.

(Smalltalk Compiler-Compiler) [BLGD17] relying on the Solidity grammar specification to build our parser. However, the Solidity grammar shows many irregularities and ambiguities that we had to tackle before creating a functional parser on SmaCC. Our process to adapt the grammar could aid other developers when dealing with similar grammatic problems.

The remainder of the paper is organized as follows. In Section 2, we present some basic concepts on parser generators and the Solidity language. Section 3 describes the parser creation and the process we used to convert the Solidity grammar into a more regular form. In Section 4, we discuss some of the parser limitations and characteristics. Section 5 describes the threats to validity for this research. Section 6 presents the related work. Finally, Section 7 presents the conclusions and outlines future work ideas.

2. Background

In this section we present all concepts needed to better understand our research. Section 2.1 presents a general classification of parsers. Section 2.2 describes tools used to build parsers. Finally, Section 2.3 presents more detail on the Solidity language and its grammar.

2.1 Parser Classification

Generally, we can classify a parser as either top-down or bottom-up. A top-down parser starts its productions from the root element of the grammar working its way down to the bottom leaves. A bottom-up parser works vice-versa, starting from the leaves and working up to the root. Moreover, either type of parser usually work on a subclass of grammars. The most common subclasses are LL(k) for top-down, and LR(k) for bottom-up parsers [ALSU06].

LL(k) denotes a type of top-down parsers where the first "L" indicates the input is read from left to right; the second "L" indicate the parser constructs the leftmost derivation for its AST; and "k" is the number of lookahead symbols used at each parsing step to choose a construction path. On the other hand, LR(k) represents a type of bottom-up parser. LR(k) constructs the rightmost derivation in reverse (represented by the "R"). The other symbols are similar to the LL(k) definition (i.e., the "L" indicates left to right input reading, and "k" the lookahead symbols) [ALSU06].

LR parser have the following advantages over other types of parsers [ALSU06]:

- If a context-free grammar can be written for a programming language, then a LR parser can most probably recognize it. Although, there are context-free grammars that are not LR, they are usually not used for typical programming languages.
- The class of grammars parsed by LR is a proper superset of LL methods. Therefore, LR grammars can describe more context-free languages than LL grammars.
- LR parser detects syntactic errors as soon as possible when reading the input.
- LR parser employ a table-driven non backtracking parsing method. For this reason, LR parsers are more efficient than recursive backtracking methods used by LL parsers.

The main disadvantage of LR parsers is that it is more difficult to implement them manually. Therefore, we need to use a parser generator to create an LR parser for a typical programming language. In practice, most LR parser generators use a LALR (LookAhead-LR) technique. LALR generates smaller tables than classical LR techniques, which improves the overall parser performance [ALSU06].

2.2 Parser Generator Tools

Creating a parser can be a difficult and time-consuming endeavor. Parser generators provide an easier way to tackle this problem [IM15]. Basically, we write the syntax specification using a formal grammar and the generator automatically builds the parser. For example, there are many parser generation tools available nowadays such as YACC [Mer93], ANTLR [Mil05], and SmaCC [BLGD17]. When we consider the Pharo Smalltalk environment, there are two prominent parser generation tools: PetitParser [BCDL13], and SmaCC [BLGD17].

We are going to give a brief analysis on some well known parser generation tools. SmaCC [BLGD17] can generate either LR(1) or LALR(1) bottom-up parsers based on a grammar specification. YACC [Mer93] is also a bottom-up LALR(1) parser generator. The original YACC was developed in C for Unix systems, but now we have implementations of it on several other languages. ANTLR [Mil05] differs from the previous tools because it is a top-down parser generator. ANTLR creates predictive LL(k) parsers. By using a variable number of "k" lookahead symbols, ANTLR can handle grammar irregularities better than LR(1) and LL(1) parsers (at the cost of performance). There are implementations of ANTLR for languages such as Java, C++, C#, JavaScript, Python, and others. However, there is no implementation of ANTLR for Pharo. Finally, PetitParser [BCDL13] is a scannerless parser generator that relies on parsing expression grammars. PetitParser is integrated with Moose, which facilitates its use on Pharo. The main problem with PetitParser is it does not read a grammar specification, and we must write the grammatic expressions using the PetitParser language.

2.3 Solidity

Solidity is a high-level contract-oriented language originally designed for the Ethereum platform. Solidity was based on JavaScript, and as such, its syntax shares many similarities [Eth17]. Solidity contracts are compiled into a specific byte code to run on the EVM (Ethereum Virtual Machine). A compiled contract can be placed into the blockchain by executing a special transaction that allocates an address to it

[BDLF+16]. This address is a 160-bit identifier that points to the smart contract. Moreover, the code for the contract resides and it is executed on the blockchain [LCO+16].

The Solidity documentation describes the current language grammar in a BNF-like format [Eth17]. Listing 1 presents a simple subset of this grammar. As we can see, a Solidity grammar starts with the *SourceUnit* rule, and it is composed by zero or more instances of a pragma directive, import directive, or contract definition (line 1). A pragma directive starts with the keyword "pragma" followed by an identifier, and then any combination of one or more characters until a semicolon (line 2). A contract definition starts with "contract", "library", or "interface" and the identifier for the definition (line 5). It is optional to use the "is" keyword and declare several inheritance specifications separated by commas (line 6). Finally, a contract must have an open curly bracket and zero or more parts (the main code for the contract) and a closed curly bracket to end its definition (line 7).

```
1 SourceUnit = (PragmaDirective | ImportDirective |
      ContractDefinition)*
2 PragmaDirective = 'pragma' Identifier ([^;]+) ';'
3
4 ContractDefinition =
5    ('contract' | 'library' | 'interface') Identifier
6    ('is' InheritanceSpec (',' InheritanceSpec )* )?
7    '{' ContractPart* '}'
```

Listing 1. Solidity Grammar Subset

Listing 2 shows a simple contract example that stores a pair of integer values. Basically, we create the contract (line 3) that persistently stores two values (lines 4-5). Moreover, the contract also has one function to set those values (lines 7-10) and another function to return the value pair (lines 12-14).

```
1  pragma solidity >=0.4.12;
2
3  contract SimplePairStorage {
4      int data1;
5      int data2;
6
7      function setData(int d1, int d2) {
8          data1 = d1;
9          data2 = d2;
10     }
11
12     function getData() constant returns (int, int) {
13         return (data1, data2);
14     }
15 }
```

Listing 2. Simple Contract in Solidity

3. Parser Creation

In this section we describe the steps we took to build our Solidity parser. We also describe our main practices to convert the grammar into a more regular form. The current version of our parser is publicly available at github[3].

3.1 Design

As we previously described (Section 2.2), there are two parser generators available in Pharo: PetitParser and SmaCC. Therefore, we had these two options to create our Solidity parser. We could also write the parser ourselves without relying on generators. However, the Solidity language specification is not stable, and a manually coded parser is more difficult to modify than a generated one.

We acquired the official grammar for the current version of Solidity (version 0.4.12, grammar document committed in 2017-06-15) [Eth17]. Although, Solidity presents a context-free grammar for its language (in a BNF-like form), this grammar is ambiguous. Moreover, the grammar was not written to be a LR(1) grammar, which is the type of grammar used by SmaCC. It is noteworthy that this grammar is not LL(1) either. Since LL(1) grammars are used to manually write parsers without generators; a hand written parser would suffer from grammar adaption problems as well. PetitParser relies on parsing expression grammar which is different from context-free grammar. Therefore, adapting the Solidity grammar to PetitParser would require more effort than to adapt it to LR(1) or LL(1). There is also a Solidity grammar specification for the ANTLR tool available. This grammar does not reflect exactly the official grammar as it shows minor adaptations for it to be used in ANTLR. However, there is currently no implementation of ANTLR for Pharo, and consequently, we cannot use ANTLR to generate our parser.

Considering that any option we decided to create our parser would require some form of grammar adaptation, we selected SmaCC as our parser generation tool. We chose to develop our parser using SmaCC for the following reasons: (i) SmaCC requires a textual context-free grammar as input that is similar to the grammar provided for Solidity; (ii) SmaCC generated parser can adapt more easily to future changes in the Solidity grammar; and (iii) SmaCC produces a LR parser, which has interesting advantages over other types of parsers (as we described in Section 2.1). Both PetitParser and a manually written parser would be more difficult to adapt and maintain than SmaCC. However, we still have to tackle the challenge to adapt the existing grammar into one that SmaCC could understand.

3.2 Converting the Grammar

In this section, we describe our main practices and lessons learned when converting the Solidity grammar to be compatible with SmaCC. We relied mostly on well know techniques for this adaptation [ALSU06]. We also used examples and solutions specific for SmaCC [BLGD17]. Our process

[3] https://github.com/RMODINRIA-Blockchain/SmaCC-Solidity, verified 2017-06-21.

and experience in converting such grammar can help other practitioners when dealing with a similar challenge.

3.2.1 SmaCC Basics

When trying to convert a grammatic specification to SmaCC, the first step is a simple textual substitution. For instance, in the Solidity grammar the "=" separates the left side of the rule from its right (i.e., the rule name from its specification). On the other hand, SmaCC uses ":" instead. Moreover, every syntactic production on SmaCC must end with a semicolon. Therefore, the first step anyone has to do when using SmaCC is to convert the production rules accordingly.

> **Lesson Learned:** Modify the grammar productions to the form:
> SyntacticRule : Prodution1 |... |FinalProduction ;

3.2.2 Put Lexical Patterns in the Scanner

SmaCC separates its specification for scanner and parser. Even though this is a classical approach with many advantages [ALSU06], this is also a limitation for SmaCC as it does not support lexical patterns in syntactic rules. Lexical patterns are easy to identify as most of them use some notation (usually brackets) to define a range among valid characters. When using SmaCC, such patterns should be placed in the scanner. It is also noteworthy that lexical patterns are handled much faster by the scanner than the parser.

In the Solidity grammar, there were many lexical patterns defined as syntactic rules. Listing 3 shows some lexical pattern examples we found in the Solidity grammar. Identifier for the Solidity language should start with a letter or an underscore or a dollar sign, followed by zero or more occurrences of letter, digit, underscore or dollar sign (line 1). A hexadecimal number starts with "0x" followed by one or more digits or characters between "A" to "F" (line 2). Finally, a decimal number is one or more digits (line 3).

```
1 Identifier = [a-zA-Z_$] [a-zA-Z_$0-9]*
2 HexNumber = '0x' [0-9a-fA-F]+
3 DecimalNumber = [0-9]+
```

Listing 3. Lexical Patterns in Solidity grammar

We placed these and all other lexical patterns we identified into the scanner part of SmaCC. Listing 4 shows the same lexical patterns showed in Listing 3 but converted to SmaCC. As we can see, it requires very little adaptation to place a lexical pattern into the scanner.

```
1 <Identifier> : [a-zA-Z_$] [a-zA-Z_$0-9]* ;
2 <HexNumber> : 0x [0-9a-fA-F]+ ;
3 <DecimalNumber> : [0-9]+ ;
```

Listing 4. Lexical Patterns converted to SmaCC

> **Lesson Learned:** Identify all lexical patterns and place them in the scanner.

3.2.3 Whitespace as Token

Usually, whitespaces are ignored by parsers as they have no syntactic value; but they are used by the scanner as delimiters to separate tokens. On the other hand, there are programming languages that uses whitespaces as a part of its syntactic structure (e.g., Python, Whitespace, etc.). That is not the case in the Solidity language, which clearly specifies in its documentation that whitespace are used as tokens delimiters [Eth17]. However, the grammar may not be aware of that, since it uses a single space in one of its rules. Listing 5 shows the number literal rule for Solidity, where it is composed of either a hexadecimal or decimal number (line 1) and optionally followed by a single space character and a number unit (line 2).

```
1 NumberLiteral = ( HexNumber | DecimalNumber )
2       (' ' NumberUnit)?
```

Listing 5. Syntactic rule using Whitespace

SmaCC has special features to handle whitespaces when they have syntactic value. However, this is not the case for the Solidity language because whitespaces should be token delimiters. If we write the rule as it is in SmaCC, the whitespace will became a keyword and it will be considered a regular token for all syntactic rules. In other words, if you put a whitespace in a syntactic rule, they will not be ignored by the parser anymore. Trying to transform the single space into a valid token recognized by the scanner would result in the same problem. Therefore, we should not use whitespaces in the syntax when they have no syntactic value.

For the scanner to ignore whitespaces (including spaces, tabs, and line feeds) we must place a lexical pattern for it with the name "whitespace". Now that the scanner ignores whitespaces, we can modify the syntactic rule by removing the space. Similarly, the grammar adapted for ANTLR shows the same change on this syntactic rule, i.e., removing the single space. This changes the grammar while it still supports the same language constructs. Even though there was only a single space before the number unit in the syntactic rule, the official Solidity compiler[4] supports any number of whitespaces before the number unit. Therefore, this grammar change did not impact the Solidity language recognized by our parser as it parses the same constructs as the official compiler.

Listing 6 shows the modified rule, since we placed *HexNumber* and *DecimalNumber* in the scanner (Section

[4] We used Remix to verify the behavior of the official Solidity compiler. Remix is an IDE and runtime environment for Solidity and it is integrated with the committed versions of the Solidity official compiler. As stated in the Solidity documentation: "the best way to try out Solidity right now is using Remix"[Eth17].

3.2.2), we adapted accordingly. The Listing also shows the scanner pattern to ignore whitespaces.

```
## Scanner Part
<whitespace>: \s+ ;

## Parser rules
NumberLiteral : ( <HexNumber> | <DecimalNumber> )
     NumberUnit? ;
```

Listing 6. SmaCC rule without Whitespace

> **Lesson Learned:** Do not use whitespaces as keyword in the parser for languages that consider them delimiters.

3.2.4 Shift-Reduce Problems

Shift-reduce is a process to handle input on bottom-up parsers, used by LR and LALR methods. However, there are context-free grammars where the shift-reduce process cannot decide if it should employ shift or reduce, thus creating a shift-reduce conflict [IM15, ALSU06]. SmaCC uses this process, and one of its error messages is a shift-reduce conflict [BLGD17].

We learned that the Solidity grammar had many shift-reduce problems for its productions. For example, the classical dangling-else problem[5][ALSU06]. Although it is, usually, possible to modify the grammar to resolve such problems, most parsers implement a different solution. In the case of the dangling-else, the most common solution is to associate "else" with the closest "if". Since these conflicts are somewhat common when working with grammars, SmaCC offers directives to handle shift-reduce conflicts.

First, we need to identify the rule and which part of it is causing the shift-reduce conflict. Unfortunately, the message provided by SmaCC on the conflict might be difficult for a beginner to understand (Figure 1).

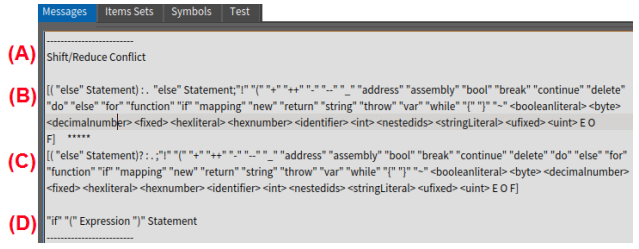

Figure 1. SmaCC Shift-Reduce Conflict Message

Based on the error message (Figure 1), we can identify the problem. Basically, SmaCC is stating which type of conflict occurred, shift-reduce in this case **(A)**. Then it shows the two productions that caused the conflict **(B)** **(C)**, i.e., where the parser could not decide which one to use. The last line shows the production until the place that encountered the problem **(D)**. Therefore, the last line shows that we have a problem with the rule after its "if (Expression) Statement" production. There is only one rule that has this production (Listing 7), and both conflicts seem to start around the "else" keyword (the classic dangling-else problem).

```
IfStatement : "if" "(" Expression ")" Statement
     ("else" Statement)? ;
```

Listing 7. Solidity If rule written in SmaCC

We have identified the problematic rule (*IfStatement*) and what part of it is most likely to be causing the conflict (*else* keyword). Now, we can resolve the conflict in SmaCC by using the directives "%left" or "%right" followed by the keywords or operators that are causing the ambiguity. The "%left" directive denotes that SmaCC should resolve the left side first when the keyword is encountered, in fact, SmaCC will employ a reduce operation. Similarly, the "%right" directive gives preference to the right side by performing a shift operation. Since we want the "else" to be paired with the closest "if", we use the "%right" directive (Listing 8). If we used the "%left" directive, the else would be associated with the farthest "if".

```
%right "else";
IfStatement : "if" "(" Expression ")" Statement
     ("else" Statement)? ;
```

Listing 8. If rule with SmaCC directive

We experienced similar conflict problems related to function modifiers in Solidity. More specifically the "constant" and "internal" keywords caused shift-reduce conflicts. We corrected them by simply using the %left directive.

> **Lesson Learned:** First identify the cause of the shift-reduce conflict. Then use either %left or %right to resolve it.

3.2.5 Expression Ambiguity

The Solidity grammatic specification for expressions is ambiguous. This is understandable because an ambiguous grammar may provide a shorter specification which is also easier to comprehend, specially for expressions [ALSU06]. Unless a parser outputs several possible ASTs (which is unlikely), the parser itself must find someway to resolve the syntactic ambiguities to build one deterministic AST. This is true for any type of parser and it is not a particular case for SmaCC.

Listing 9 shows an edited sample of the expression in Solidity. For instance, consider the following expression: 1+2*3. According to the syntactic rule, two possible ASTs can be built for this expression (Figure 2). Since there is

[5] Dangling-else is problem that occurs on parsers when an optional "else" clause in nested "if" statements becomes ambiguous.

more than one AST for the same input, this rule and, consequently, the grammar is ambiguous.

```
Expression = ...
   | Expression ('*' | '/' | '%') Expression
   | Expression ('+' | '-') Expression
   | ...
```

Listing 9. Solidity Ambiguous Expression

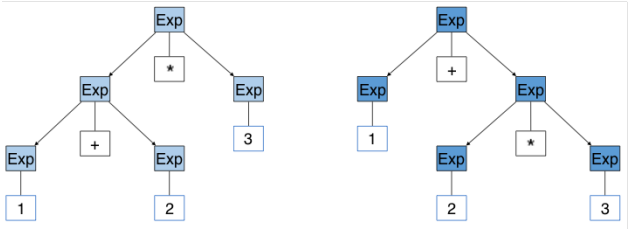

Figure 2. Ambiguous Expression AST

When an ambiguous grammar such as this expression (Listing 9) is placed on SmaCC, the tool outputs error messages of shift-reduce. In this case, we can either rewrite the grammar to an equivalent unambiguous grammar (which may require a lot of expertise), or we can use directives to resolve the conflict. The directives are the same ones we used for the shift-reduce problems (Section 3.2.4), and we use %left and %right on the expression operators to resolve the ambiguity. There is one additional caution when using this directives for expressions, we need to observe the operators precedence. The lowest precedence operator should be the first directive [BLGD17]. Listing 10 shows the directives we used for the expression example.

```
%left "+" "-";
%left "*" "/" "%";
Expression
   : #... other derivations
   | Expression ("*" | "/" | "%") Expression
   | Expression ("+" | "-") Expression
   | #... other derivations
   ;
```

Listing 10. Solidity Expression in SmaCC

Lesson Learned: Use %left or %right directives for the expression operators to resolve its conflicts, paying attention to their precedence order (lowest comes first).

3.2.6 Same Symbol as Separator and Operator

Another interesting challenge we found to parse Solidity was that the language uses the same symbol (comma) as a separator for expression lists but also as an operator for the expression itself. Listing 11 shows a edited sample of these rules in the Solidity grammar. This causes a serious problem because when the parser finds a comma in the input it does not know if it is an operator for the current expression (matching the Expression rule) or a separator to the current expression and the beginning of a new one (matching ExpressionList). This is a potential problem for any parser due to the ambiguity of matching either rule when encountering a comma.

```
ExpressionList = Expression ( ',' Expression )*
Expression = ...
   | Expression? ( ',' Expression )
   | ...
```

Listing 11. Solidity Expression List and Expression

Specifically in SmaCC, this type of problem generates reduce-reduce conflicts and they are more complex conflicts to resolve [BLGD17]. A reduce-reduce problem arises when the shift-reduce parsing method has several possible reductions to make and cannot decide which one to use [ALSU06]. There is no directive or easy solution to handle reduce-reduce conflicts, we must get to root of the problem and change the grammar. We already know the conflict arises on the comma being used as either an operator and separator. Therefore, we looked in the Solidity documentation for examples of comma used in expressions. Listing 12 shows an edited example, the comma is used to define tuples in Solidity expressions, which is important when a function returns multiple values.

```
contract CommaExample {
  uint[] data;

  function f() returns (uint, bool, uint) {
    return (7, true, 2);
  }

  function g() {
    // Declares and assigns the variables.
    var (x, b, y) = f();
    // Assigns to a pre-existing variable.
    (x, y) = (2, 7);
    // Common trick to swap values
    (x, y) = (y, x);
    // Components can be left out (also for
        variable declarations).
    (data.length,) = f(); // Sets the length to 7
    // The same can be done on the left side.
    (,data[3]) = f(); // Sets data[3] to 2
    // Components can only be left out at lhs...
    // ... with 1 exception
    (x,) = (1,);
  }
}
```

Listing 12. Solidity Comma usage Example

As we can see in Listing 12, tuples are always within parentheses. We searched for other examples of tuple usage and the ones we found also followed this guideline (i.e., every tuple were enclosed by parentheses). Adding parentheses

to the tuple expression would resolve the ambiguity with expression list. Moreover, we verified tuples expressions on the official Solidity compiler, and we discovered that it supports tuples only when they are enclosed by parentheses. Therefore, we changed the grammar by only allowing tuples inside parentheses as well. Listing 13 shows the SmaCC specification focusing only on tuples expression and expression list. This new specification generates the same productions as before with the exception that tuples must now be inside parentheses. Unfortunately, this is a limitation we introduced in our parser in order to resolve the conflict. On the other hand, such limitation is also present in the official Solidity compiler, and thus, our parser is recognizing the same constructs.

```
ExpressionList
   : Expression ("," Expression)*
   ;
Expression
   : #... other derivations
   | "(" Expression ("," Expression? )+ ")"
   | "(" ("," Expression?)+ ")"
   | #... other derivations
   ;
```

Listing 13. SmaCC Expression List and Expression

Lesson Learned: Using the same symbol as a separator and operator can cause a serious ambiguity problem in a form of reduce-reduce conflict that can only be solved by rewriting the grammar.

3.3 Parser Evaluation

To test our parser functionality, we verified if it can recognized smart contracts written in Solidity. Even though we adapted the Solidity grammar to use SmaCC, the important concern is to see if we can parse the same language. We used Etherscan[6] to acquire Solidity smart contracts that were made publicly available with its source code. Etherscan possess a library of more than 2.7K verified smart contracts. Table 1 shows the number of contracts by Solidity version in Etherscan. Table 1 also shows the number of distinct contracts by version, because there can be more than one version of the same contract in the Etherscan database.

Version	# of Contracts	# of Distinct Contracts
0.1.x	10	10
0.2.x	73	64
0.3.x	552	323
0.4.x	2,096	1,175
Total	2,731	1,572

Table 1. Ehterscan Contracts by Solidity Version

[6] https://etherscan.io/contractsVerified, verified 2017-08-01.

We selected only distinct contracts with version 0.4.x for a total of 1,175 contracts. Since we designed our parser using the Solidity version 0.4.12 (the most current version at that time), contracts from version 0.4.x should be parsable with it. We created a random sample of 117 from the 1,175 contracts (version 0.4.x) to test the parser. We successfully parsed all 117 smart contracts from our random sample.

4. Discussion

In this section, we present a brief discussion on our implemented Solidity parser.

4.1 Parser Limitations

Although we were successful in building a Solidity parser running on Pharo, we did identified a few limitations in our implementation. These limitations were caused by the grammatic transformations we made to tackle irregularities in the language.

The main limitation was caused when we changed the grammar to avoid a major ambiguity problem on expression lists and tuples expression (Section 3.2.6). We were unable to come up with a better solution that did not impact the productions. Thus, our parser only allows tuples inside parentheses. This is a limitation that did not exist in the original grammar (Solidity version 0.4.12, grammar document committed in 2017-06-15). However, even the official Solidity parser presents the same limitation. We can argue that either the Solidity parser is not up to date with the grammar, or that the grammar is inconsistent with the implemented parser. Presumably, the official parser should be the primary reference for what is accepted in the Solidity language, which would indicate that it is not a limitation but the actual intended behavior.

4.2 Grammar Changes

In this paper, we described grammar changes and adaptations to create a Solidity parser using SmaCC. Even tough, the resulting grammar is not exactly the same as the original, our main goal was to achieve an equivalent grammar that can describe the same Solidity language. We claim that our parser recognizes the same language based on the following reasons: (i) the limitations acknowledged in our parser are also present in the official Solidity compiler, which indicates that is the current accepted behavior for contracts written in Solidity; and (ii) we successfully parsed 117 verified contracts that were deployed in a real blockchain platform.

4.3 AST Generation

In this paper, we did not describe the construction of the AST. The focus of this paper is the parser and the grammatic changes necessary to handle Solidity. We believe that AST generation would deviate from our intended focus. However, we feel it is noteworthy to mention that our parser generates a complete AST as the main output of its parsing process.

With this AST we can now develop tools more easily to improve our integration support with Solidity.

5. Threats to Validity

In this section, we identify and classify the threats to validity in our experimental process.

5.1 External Validity

In this paper, we analyze and modify the Solidity grammar for it be compatible with SmaCC. We cannot claim that our proposed solutions could be generalized to any grammar. However, we tried to mitigate this threat by explaining the process and techniques used to facilitate the adaptation to other grammars. Moreover, this paper focuses only in our experiences handling Solidity, and analyzing other grammars would fall outside the scope of this research.

5.2 Internal Validity

We relied on our expertise to convert the Solidity grammar for it to be used on SmaCC to generate a functional parser. Therefore, if different people participated in the conversion process, the grammatic adaptations and transformations may also be different. We tried to mitigate this threat by relying mostly on well known techniques for resolving conflicts on general LR parsers [ALSU06] and SmaCC [BLGD17]. Therefore, if other people also relied on those techniques the results would be similar.

5.3 Conclusion Validity

Even tough we created a parser to interpret contracts written in Solidity by using SmaCC, we cannot guarantee that another parser generation tool would provide similar or better results. This remains an open question and a possible future work idea.

6. Related Work

We divided the related work into two main categories: (i) parsing conflicts; and (i) Solidity smart contracts.

6.1 Parsing Conflicts

Isradisaikul and Myers [IM15] describe the difficulty in solving grammar conflicts with LALR parser generators. They propose an algorithm that generates better counterexamples for LALR parsers. A counterexample shows a parsing scenario that causes an ambiguity conflict in the LR method. Such scenarios helps developers to diagnose the conflict's source and identify problems in the grammar specification. The authors implemented their algorithm in Java as an extension to the CUP parser generator and they also evaluated their approach in 20 grammars against a state-of-the-art ambiguity detector. Their algorithm usually finds more counterexamples in less time than the compared techniques.

Passos et al. [PBB07] proposes a methodology to resolve conflicts in the LALR parsing method. The authors describe the challenges for handling conflicts without changing the defined language. They used YACC to generate a parser for the Notus language to illustrate the difficulty to resolve conflicts for LALR(1). The authors created a tool, called SAIDE, based on their methodology which is an improvement over the regular methods available to handle conflicts in LALR parsers.

6.2 Solidity Smart Contracts

Most research related to Solidity smart contracts leans towards security. For instance, Luu at al. [LCO+16] try to makes Ethereum based contracts smarter and more secure. First the authors analyze possible exploits that someone can use to take advantage, and then they propose solutions to lower the vulnerabilities in the platform. They also build a tool written in Python, called Oyente, that flags potential security risks in smart contracts. Their tool analyze contracts on the EVM byte code instead of Solidity, because according to them most contracts does not have their Solidity source publicly available. The authors evaluate their tool using 19K contracts on a quantitative and qualitative level. Their tool was successful in detecting security risks in 8K of the evaluated contracts.

Bhargavan et al. [BDLF+16] propose a framework to analyze the security and correctness of contracts running on Ethereum. The authors create a functional language, called F*, and translate both Solidity and EVM byte code contracts to it. According to the authors, F* is easier and better to verify the contracts. For the evaluation, the authors acquired 396 contracts but were only able to translate 46 to F*. The authors acknowledge that their work is a preliminary stage, and its results were good enough. The authors also conclude that static analysis tools might be easier to employ for Ethereum smart contracts.

7. Conclusion

Blockchains and smart contracts are a prominent field that has attracted much attention in recent years. Solidity is a major language used to write smart contracts and it is supported by many blockchain platforms.

In this paper, we developed a Solidity parser by using SmaCC. Developers can use this parser to increase the integration between Solidity and Pharo by implementing more tool support. We tested 117 smart contracts that were successfully interpreted by our Solidity parser. Even tough, our parser has limitations; it is still parsing the same language constructs when compared to the official Solidity compiler. Therefore, we claim that our parser recognizes the Solidity language. As far as we know, this is the first Solidity parsing tool available for Pharo Smalltalk.

We also showed many practices and lessons on how to convert a real language grammar to a more regular form that an LR parser can work with. These lessons may aid other

developers when using SmaCC or other parser generators with similar characteristics.

We have the following ideas for future work: (i) improving the parser by addressing its limitations; (ii) build a semantic analysis for the parser; (iii) compare SmaCC against another parser generation tool; (iv) develop more tools for Solidity contracts; (v) recommendation system for Solidity to suggest good and secure practices when writing smart contracts.

Acknowledgments

Our research is supported by UTOCAT.

References

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[BCDL13] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. Petitparser: Building modular parsers. In *Deep into Pharo*, chapter 18, pages 377–411. Square Brackets Associates, 2013.

[BDLF+16] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *2016 ACM Workshop on Programming Languages and Analysis for Security*, PLAS '16, pages 91–96, New York, NY, USA, 2016. ACM.

[BLGD17] John Brant, Jason Lecerf, Thierry Goubier, and Stéphane Ducasse. Smacc: a compiler-compiler, 2017.

[DAK+15] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, Andrew Miller, and Elaine Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. Cryptology ePrint Archive, Report 2015/460, 2015. http://eprint.iacr.org/2015/460.

[DAKM15] Kevin Delmolino, Mitchell Arnett, Ahmed Kosba, and Andrew Miller. A programmer's guide to ethereum and serpent. Technical report, University of Maryland, Computer Science Department, Ethereumlab, 2015.

[Dzi15] Stefan Dziembowski. Introduction to cryptocurrencies. In *22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1700–1701, New York, NY, USA, 2015. ACM.

[Eth14] Ethereum Foundation. Ethereum's white paper., 2014.

[Eth17] Ethereum Foundation. Solidity documentation release 0.4.12. Technical report, 2017.

[HL16] Adiseshu Hari and T. V. Lakshman. The internet blockchain: A distributed, tamper-resistant transaction framework for the internet. In *15th ACM Workshop on Hot Topics in Networks*, HotNets '16, pages 204–210, New York, NY, USA, 2016. ACM.

[IM15] Chinawat Isradisaikul and Andrew C. Myers. Finding counterexamples from parsing conflicts. In *36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 555–564, New York, NY, USA, 2015. ACM.

[LCO+16] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, 2016. ACM.

[LMH16] Benjamin Leiding, Parisa Memarmoshrefi, and Dieter Hogrefe. Self-managed and blockchain-based vehicular ad-hoc networks. In *2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*, UbiComp '16, pages 137–140, New York, NY, USA, 2016. ACM.

[LTKS15] Loi Luu, Jason Teutsch, Raghav Kulkarni, and Prateek Saxena. Demystifying incentives in the consensus computer. In *22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 706–719, New York, NY, USA, 2015. ACM.

[Mer93] Gary Merrill. Parsing non-lr(k) grammars with yacc. *Software Practice and Experience*, 2(8):829–850, 1993.

[Mil05] Ashley Mills. Antlr tutorial, 2005.

[Nak09] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system., 2009.

[PBB07] Leonardo Teixeira Passos, Mariza A. S. Bigonha, and Roberto S. Bigonha. A methodology for removing lalr(k) conflicts. *Journal of Universal Computer Science*, 13(6):737–752, jun 2007. http://www.jucs.org/jucs_13_6/a_methodology_for_removing.